Scribe notes for 11/11/14
by Diane Yang, Kara Shen, and Chris Lim

Process control

Game plan:

- fork
- exec
- pipes

All code can be found in the l19 directory

Fork

`fork1.c`

Recall that fork returns 0 to child and returns **child's process ID** to parent

- Thus, we know that the child's process ID cannot be 0
- We also know it cannot be negative
- Note that a child's PID can be less than its parent's

Race condition: when behavior of a program depends on scheduling

- Often the cause of bugs. Hardest to debug when the bug only surfaces rarely
- **nfork()** is useful for debugging because it makes these bugs appear more often
    - Nondeterministically causes the parent or the child to run first.
- **fork()**, in contrast, causes the parent to run first
    - (note: only because it's more likely the parent runs first since the function returns to the parent–there is still a chance that as soon as the return happens, the scheduler intervenes and runs the child!)

`nfork.h`

- Nondeterministically runs child or parents first
    - Doesn't change behavior of fork, just makes nondeterminism more obvious for us
- Why does the parent run first when use the normal fork?
    - Normally a system call always returns to the process that made the system call
    - The only way that we could get the printf from the child first is if the timer interrupt in a very specific place
- So, to prevent program from crashing when this sometimes does happen, need to program pretending that it will happen all the time

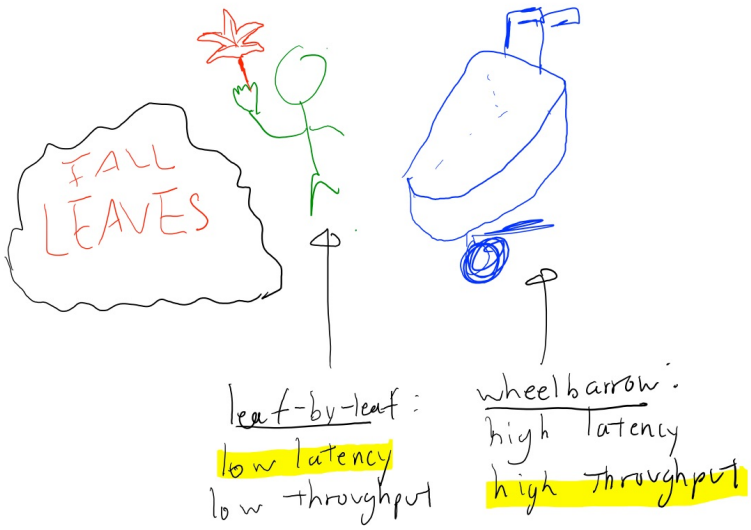
`fork2.c`

- This code will create 4 processes, print 4 lines

`fork2b.c`

- This code will also create 4 processes
- BUT it will output 7 lines when written to a file.
    - Why?
        - First prints a line to standard output, then forks twice
        - When the standard io library notices that the output is directed to terminal, it doesn't buffer anything, it is written to the console

- When the standard io library detects that the output is directed to a disk file, it privileges throughput over latency
    - **Latency**: speed it takes to get anything done
    - **Throughput**: speed it takes to get a bunch of work done
    - leaf analogy: one single leaf would have small latency but low throughput too
    - Ideally, we want **low latency** and **high throughput**!



- So stdout hooked up to console → low latency mode
- stdout hooked up to file on disk → high throughput mode

`forkmix.c`

- When you run it, you see 10,000,000 lines - a block of “baby”s and a block of “MAMA”s
    - Fork buffers its output
- When you pipe it into a file, you would expect to see exactly 10,000,000 lines.
- However, you only see *roughly* 10,000,000 because of a race condition in Linux (this is intentional, because it’s faster).
- You would see exactly 10,000,000 on a Mac, for example.
- **uniq x** - takes out consecutive duplicate lines except the first of each set of duplicates
- We find that **uniq x** (after having done ./forkmix > x) yields strange output! Why?
    - Because our strings are not a power of 2, and size of each buffer is 4096!

`forkmix2.c`

- Outputs 5,000,000 characters instead of 10,000,000 characters
- Why? See below!

More on the processes/forking to explain forkmix2.c

Shell process opens a file x

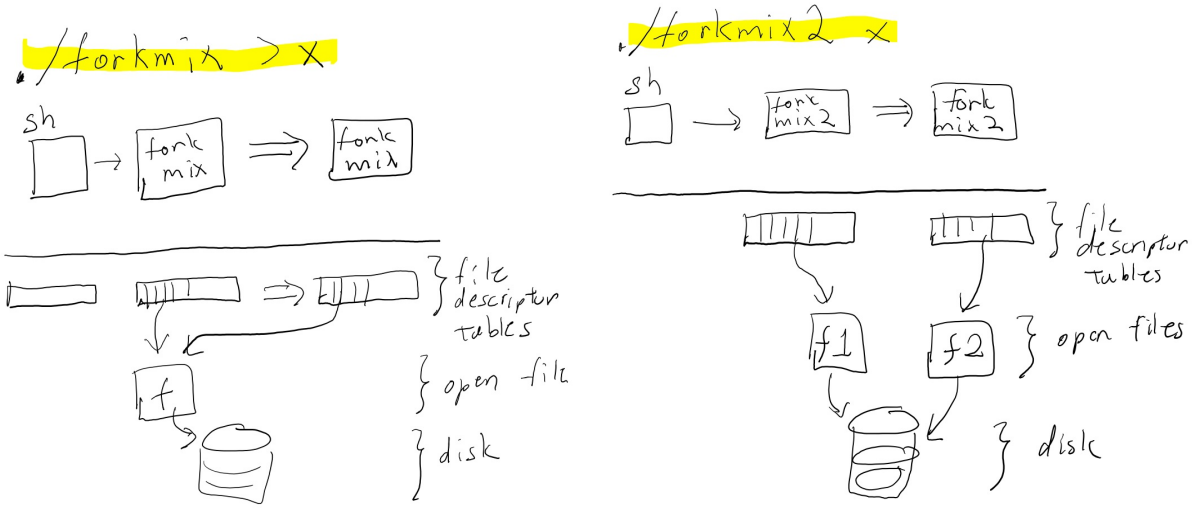
- There is a disk that contains data for the file
- There is a file structure that represents the open file

When shell runs forkmix

- Creates forkmix program whose file descriptor table points to the file descriptor (slot 1)
- Forkmix then forks
    - Makes copy of file descriptor table, but does not copy the open file structure
    - Parent and child are sharing the open file description (but they have different file descriptors)
- Writes not meant to collide

compare: when shell runs forkmix2

- File descriptor table
- Forked to created forkmix2
- Calls fopen(x)
    - Each has different open file description (both point to disk)
- Each process has different offset into the file, and they both start at 0
    - Result: overwrites file completely
    - Outputs 5,000,000 characters (instead of 10,000,000 characters)



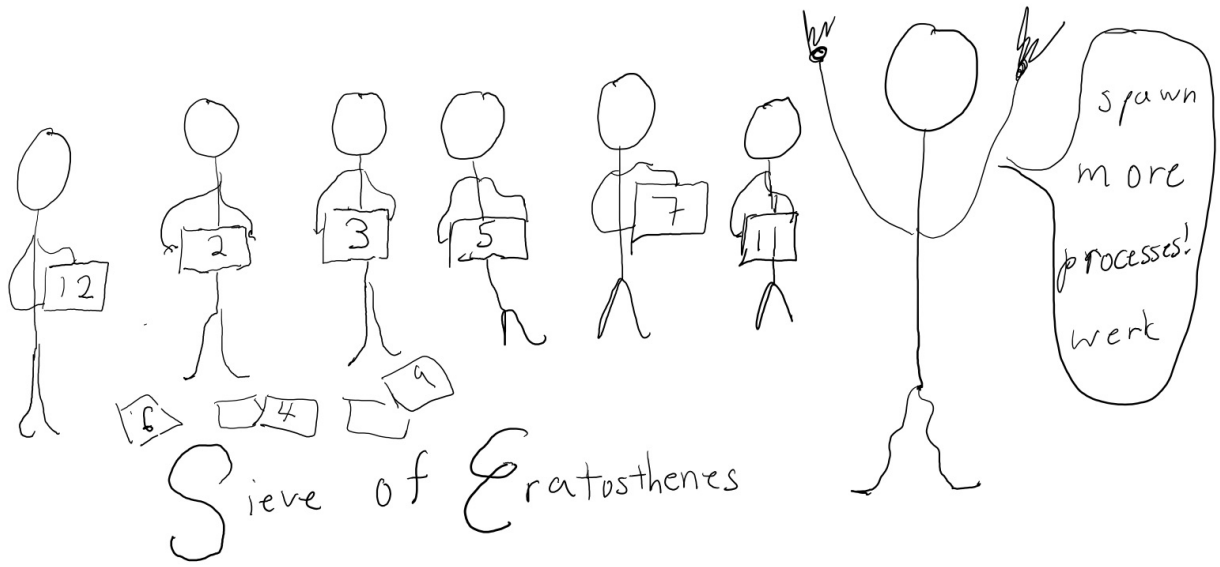
Exec

`myecho.c`

- myecho shell
    - Prints “about to exec…”
    - Call execv
    - Prints “finished exec…”
- But the finish line printf is missing!
    - Shell replaced itself with myecho
    - Only way that the the last line is printed is if exec fails
        - exec only returns -1 or never returns

`runmyecho.c`

Sieve of Eratosthenes



- Algorithm to find prime numbers
    - Given a stream of natural numbers, every time you see a new prime, ignore all of its multiples in the rest of the stream
- We can model this using Unix processes!
    - seq program gives numbers in range
    - filtermultiples prints numbers that are not multiples
- seq 2 100 | ./filtermultiples 2 | ./filtermultiples 3 | ./filtermultiples 5…

Pipes

- How to make the above command line happen automatically?
    - **pipe**: system call for creating a pipe
        - int pipefd[2];
        - int r = pipe(pidpefd);
        - pipefd[0] = READ END
        - pipefd[1] = WRITE END
    - When a process calls pipe, creates a new file structure in the kernel, opens two file descriptors (one to the write end, one to the read end)
        - Data written to the write end is readable from the read end

`pipedemo.c`

- Opens a new pipe
- Forks
    - Copies file descriptor table
    - Copies descriptors to the pipe
- Child writes from the write end
- Parent reads from the read end

More on pipes:

- When does a pipe indicate end of file?
    - Pipe reads end of file when all write ends to the file are closed
    - Pipes are associated with a buffer in the kernel
    - Allows write end to generate a bunch of data before read end is accessed
    - **remember:** close both parent and child write ends of pipe to receive EOF signal on read end!
- How to figure out how big the buffer is?
    - Don't read from the read end
    - Keep writing, and eventually the writer will block

`pipesizer.c`

- Opens a pipe
- In a loop, writes one character to the pipe
- Stops until the buffer is full
- 64 KB

`primesieve.c`

- On each step, create a new child process by calling pipe and then fork
    - Call pipe every time because number of pipes is proportional to the length of the sieve
- Parent process stands at the end, receiving new primes
    - When receives a new prime, calls and new pipe, new child, forks the child
- dup2 - take file descriptor that isn't 0 or 1 and moves it so that it will be 0 or 1
- First child: runs seek
    - All later children: run filtermultiples
- Reset parent process to read from the last injected
- Read next prime from standard input
- Print the prime, go around the loop
- LOOK AT CODE YOURSELF!

Donald Knuth article

- The power of pipes: can find the 10 most common words in a long article using only 6 shell programs + pipes